Flexible Audio Player

A Project by

Jason Franco S. Conanan
Miguel Emmanuel D.J. Suarez
Nico Angelo F. Villarica

Submitted to

Luisito L. Agustin
Instructor, ELC152

In Partial Fulfillment of the Requirements for the Course
ELC152: Signal Processing

Department of Electronics, Computer and Communications Engineering
School of Science and Engineering
Loyola Schools
Ateneo de Manila University
Quezon City, Philippines

October 2010

Abstract

A console-based flexible audio player is implemented using C++. Along with this is the design and implementation of an audio class that contains functions for playback, reverse playback, and pause and resume. The flexible audio player is capable of playing 8-bit, 16-bit, 24-bit, or 32-bit single- and dual-channel WAV audio, with sampling rates of up to 44.1 kHz.

Acknowledgements

We would like to acknowledge the following people who made significant contributions to this work:

First and foremost, we would like to express our thanks to our God, who is the source of all Wisdom and Knowledge.

Thanks to Mr. Luisito Agustin for his unwavering guidance as we completed this project.

Thanks to our classmates for their support and suggestions which helped in the refining of our project.

Thanks to the XMCS group (Anna Veronica C. Baterina & Jeric V. Macalintal) whose source code and documentation served as a guide to our project.

Table of Contents

1. Introduction 6
    1.1. Project Objectives 6
    1.2. Project Scope 6
    1.3. Overview of Components 6
    1.4. Supported Audio File – The WAV File Format 7
2. WAV Audio Playback Implementation 9
    2.1. Forward Playback 9
    2.2. Reverse Playback 10
    2.3. Forward Playback and Reverse Playback Procedure Flowcharts 11
3. The WAV Class 13
    3.1. Constructor and Destructor 13
    3.2. Data Members 13
    3.3. Accessors 14
    3.4. File Input Function 15
    3.5. Friend CALLBACK Function 16
4. The UserInterface Class 17
    4.1. Constructor and Destructor 17
    4.2. Objects and Data Types 17
    4.3. Functions 17
5. User's Manual 19
    5.1. CDROM Contents 19
    5.2. Using the Software 19
6. Results and Recommendations 23

Appendix 1. Source Code 24
        A1.1. WAV Class 24
            A1.1.1. WAV Class Definition: *wav.h* 24
            A1.1.2. WAV Class Implementation: *wav.cpp* 26
        A1.2. UserInterface Class 34

A1.2.1. UserInterface Class Definition: *userinterface.h*.....34
A1.2.2. UserInterface Class Implementation: *userinterface.cpp*.....35
A1.3. main() function: *main.cpp*.....39
Bibliography 40

1. Introduction

1.1. Project Objectives

The objective of this project is to create an audio player that can perform forward playback and reverse playback of audio files in the .wav format. The player must also include pause and resume options during forward or reverse playback.

1.2. Project Scope and Limitations

The flexible audio player has support for 8-bit, 16-bit, 24-bit, or 32-bit WAV format audio.

Forward and reverse playback of audio is limited to single-channel and dual-channel audio. The sampling rates for forward and reverse playback ranges from 8 kHz to 44.1 kHz. Forward and reverse playback can be paused and resumed by the user at anytime during playback. However, if the user wishes to stop playback, he must the load the file again.

The audio for playback is read from a WAV audio file (has the .wav file extension).

1.3. Overview of Components

The creation of the Flexible Audio Player involves the implementation of the following classes.

- An WAV class for the main components of this project. The WAV class would provide forward and reverse playback functionalities, as well as functions for pausing and resuming, and stopping of playback in either direction.
- A UserInterface class for the user interface of the program. The UserInterface class provides the functions needed for the program's console-based interface.

The three classes above are used to create the Flexible Audio Player.

1.4. Supported Audio File – The WAV File Format

The WAV file format is a subset of Microsoft's RIFF specification for the storage of multimedia files.[1] A WAV file contains a header and the raw data chunks (in time format). Figure 1.1 shows a diagram of the canonical WAV file format.

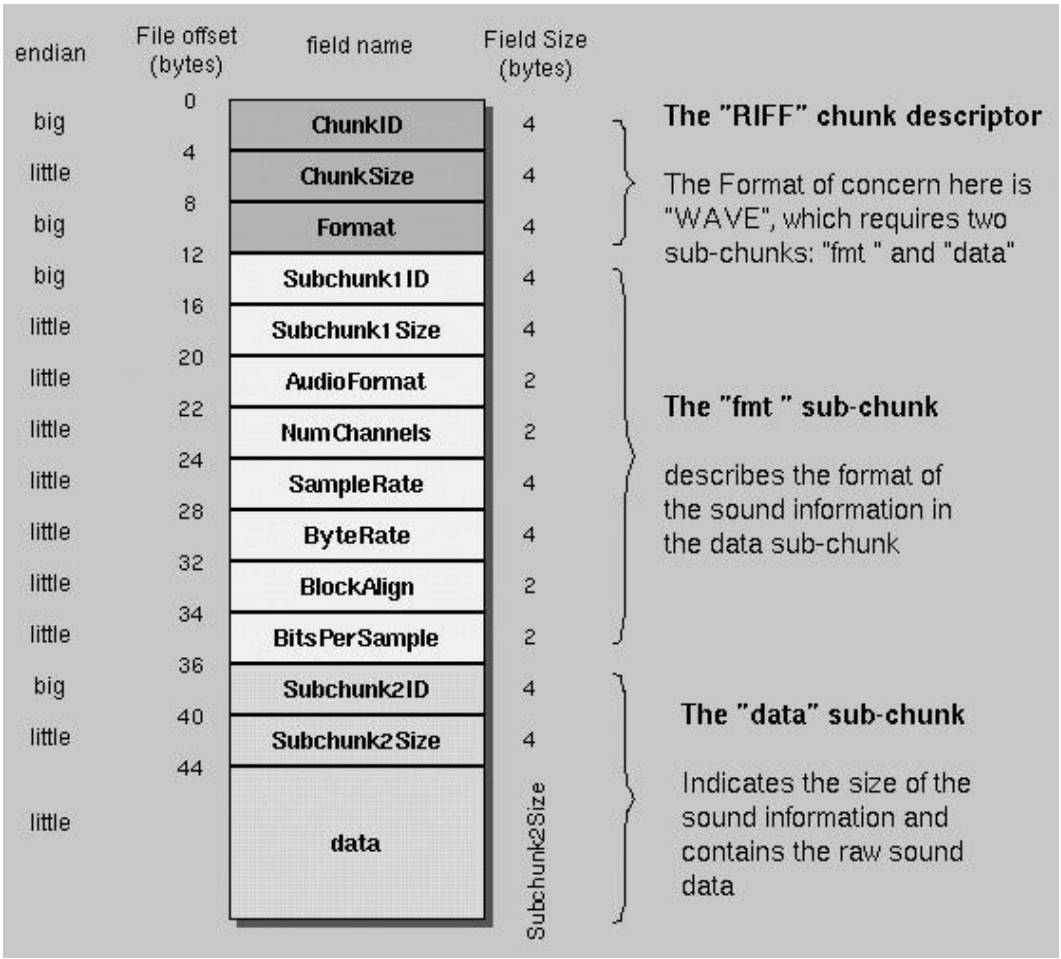

Figure 1.1. The Canonical WAV File Format

The header, the beginning of a WAV file, is used to provide specifications on the file type,

[1] Reference 6

sample rate, sample size, bit size of the file, and the file's overall length. Sample rate is the number of samples per second. When an audio file has a sampling rate of 44,100 Hz, this means that 1 second of audio has 44,100 samples. Bit size determines how much information can be stored in a file. Most WAV files today have a bit size of 16 bits. 8 bit WAV files are smaller, but have less resolution. Bit size deals with amplitude. The greater the resolution of the file, the greater the dynamic range of the file. Channels are the number of separate recording elements in the data. One channel is mono and dual channel is stereo. The data is the individual audio samples. An individual sample is the bit size times the number of channels. Samples are placed end-to-end to form the data.[2]

[2] Reference 3

2. WAV Audio Playback Implementation

This chapter provides a discussion on WAV audio playback, specifically the implementation used in this project. The implementation supports forward and reverse playback. The WAV file contains audio samples which the program divides into blocks for efficient memory usage. The blocks of audio samples are stored in a list as elements. Whether it is forward or reverse playback, these blocks are sent one-by-one to the audio output device.

2.1. Forward Playback

The concept of playback revolves around the idea of sending blocks or samples of audio data into an output audio device. A linked list is prepared that will serve as a storage for the audio data and is later sent to the audio output device one block at a time. In order to obtain these samples that will be stored in the linked list, we first have to use the *fread* function to read the contents of the wave file and then store them temporarily into a buffer. After this, the contents of the buffer are transferred to the list. When the list is filled with all the audio data samples, only then can the contents be sent to the audio output device.

First, the audio file is read, starting from the header and then the data chunks. The data obtained from the header is extracted and incorporated into the WAVEFORMATEX structure, which defines the properties of the entire audio data. After this is done, the data read from the file is copied to a buffer and then stored into a list. For playback to be achieved, first the audio output device has to be opened, and then the data within the list is sent to the audio output device and then played one at a time. But before sending the data to the device, the blocks are first prepared in multiple buffers in order to minimize the amount of time spent in between playback of one bock of data and the next. By queuing the blocks being sent to the audio device, we are effectively playing a group of samples and after they are done, a new set of queued data is ready to be played and thus

continuously provides a much more efficient method of playback. Also, using variables that compare the total number of blocks with the current blocks being processed, it is possible to determine when the playback of the entire data chunk is done. After the playback is finished, the audio output device should be closed. These steps are also followed for Reverse Playback, discussed below.

To implement the pause and resume function for forward playback, a forward iterator is used to keep track of what audio block has been played. As an audio block is played, the iterator increments. When the user pauses the playback, the iterator indicates what block will be played when the user resumes the forward playback.

2.2. Reverse Playback

The concept of reverse playback is very much similar to the concept of forward playback discussed above. A separate linked list is prepared that will serve as a storage for the reversed audio samples and these will later be sent to the audio output device one block at a time. In order to obtain these reversed samples, we first have to use the *fread* function to read the contents of the wave file and then store them temporarily into a separate buffer. The program then reverses the samples and stores them in the separate linked list. When the list is filled with all the reversed audio samples, only then can the contents be sent to the audio output device.

To implement the pause and resume function for reverse playback, a separate, reverse iterator is used to keep track of what audio block has been played. As an audio block is played, the reverse iterator increments. When the user pauses the playback, the reverse iterator indicates what block will be played when the user resumes the reverse playback.

2.3. Forward Playback and Reverse Playback Procedure Flowcharts

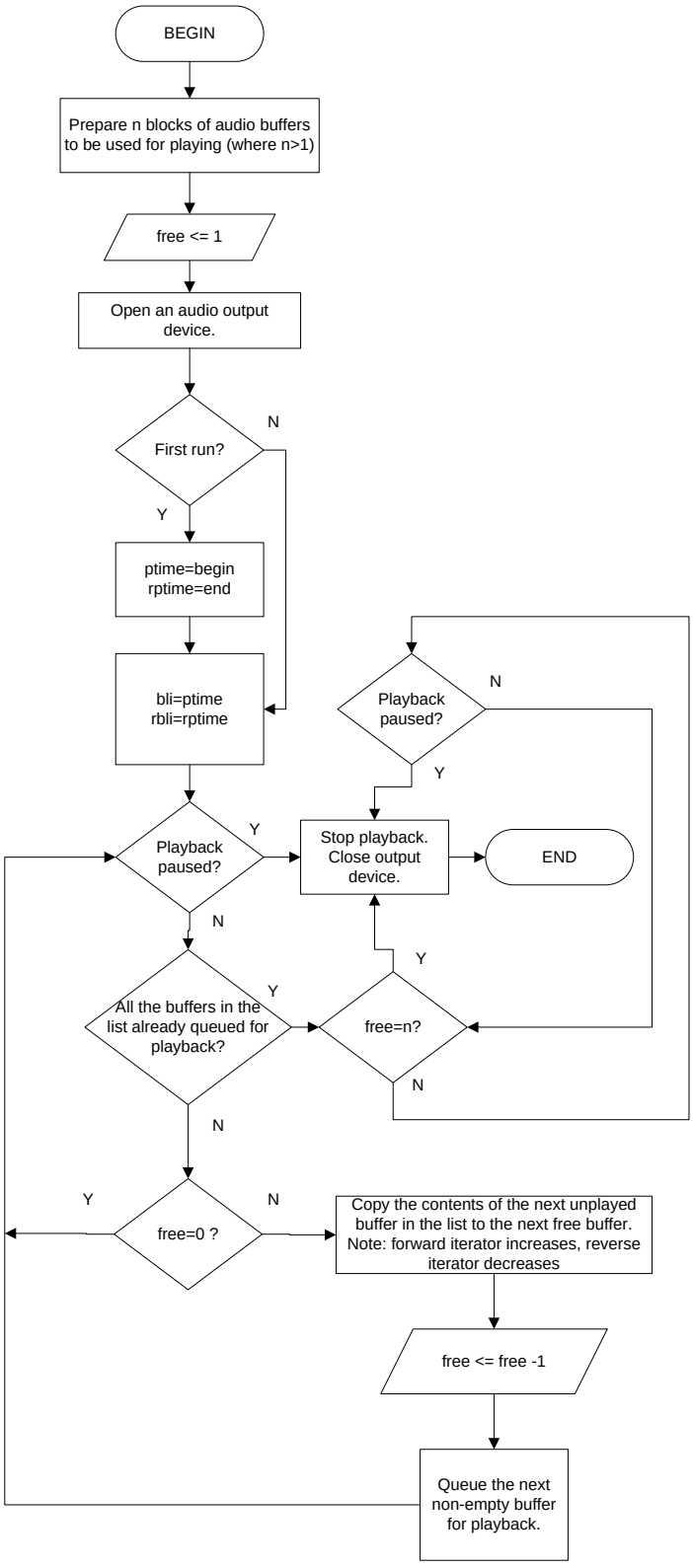

Figure 2.1. Forward Playback Procedure

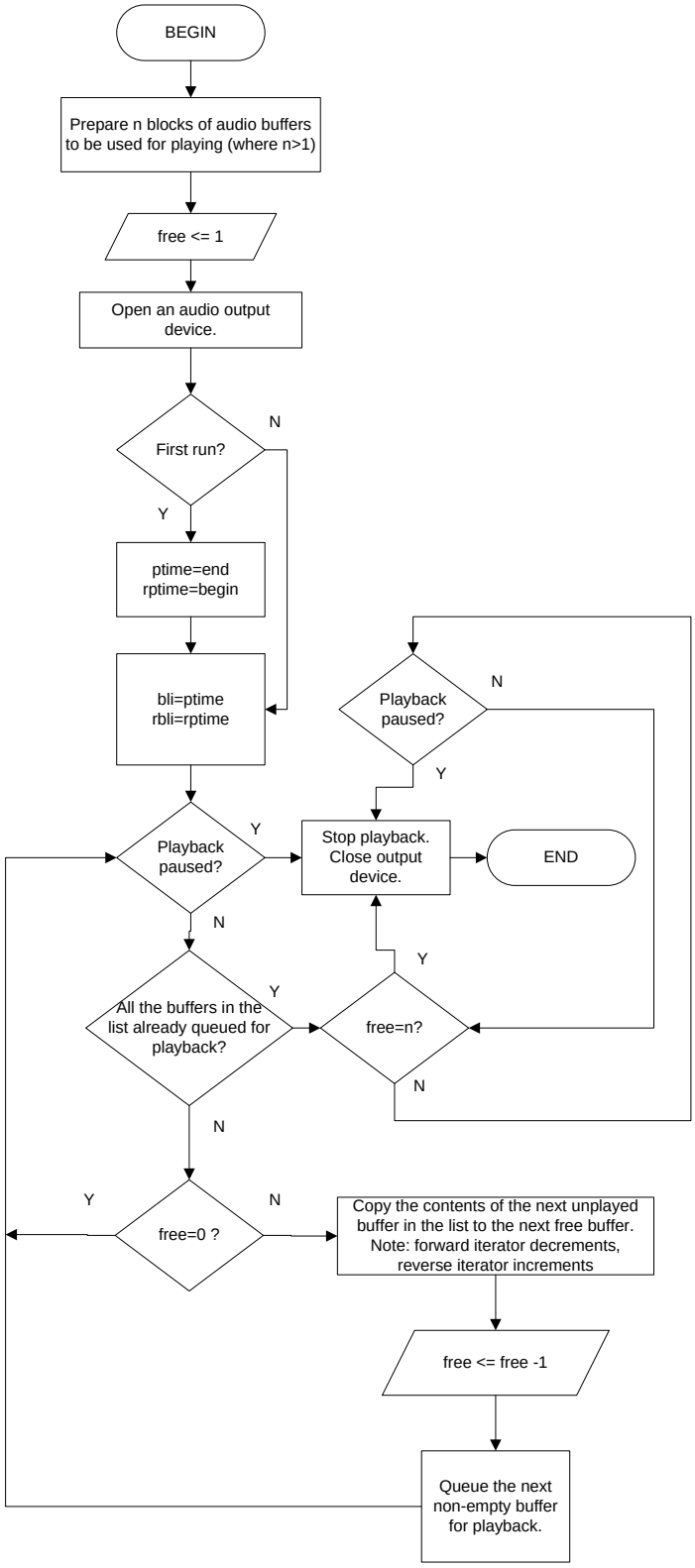

Figure 2.2. Reverse Playback Procedure

3. The WAV Class[3]

The WAV class provides the WAV forward and reverse playback, and pause and resume functionalities. It is the main class component of this project. This class was based on the XMCS class of the XMCS Player and Recorder project.

3.1. Constructors and Destructor

WAV()

The default constructor initializes the fields of the *WAVEFORMATEX* structure to their default values; some of these values are constant. It also initializes the buffer size indicators to 0.

WAV(WAV& wav)

The WAV audio is stored using a linked list, with each element containing a character pointer (character array). The copy constructor was defined so that copying of audio from another XMCS object is done properly.

~WAV()

The destructor unloads the currently loaded audio, if any, which causes any previously allocated memory for the audio samples to be deallocated.

3.2. Data Members

WAVEFORMATEX wfx

This structure is a private member that contains the format of the XMCS audio. Some of its fields are kept constant.

[3] Reference 1

list<char> audioBuffer*

This is a private member that is used to store the audio samples.

int maxBufferSize

This is a private member that is used to store the maximum buffer size in *audioBuffer*, the number of bytes in the first n – 1 elements, where n is the size of *audioBuffer*.

int lastBufferSize

This is a private member that is used to store the number of audio bytes in the last buffer of *audioBuffer*. This may be less than or equal to *maxBufferSize*.

volatile int freeBlockCount

This is a private member that is used to store information needed for audio playback. It contains the number of free buffers available. The information it provides is used to either determine whether or not a free buffer is already available or determine whether or not playback of the whole file is already done. It is modified in more than one thread of execution in the program; thus it was declared with the *volatile* keyword.

CRITICAL_SECTION criticalSection

This is used to create a critical section. The critical section is entered every time the value of *freeBlockCount* is modified since *freeBlockCount* is modified in two functions which may execute concurrently. This ensures that only one function could modify it at a given time.

3.3. Accessors

The accessors are public functions used to either read or modify the WAV information, which are stored as private data members. These functions prevent invalid values to be assigned to the fields or data types. For every field or data type that stores a WAV audio or file information, there is a corresponding pair of get and set accessors.

int GetChannels()

This function returns the number of channels, which is contained in *wfx*.

bool SetChannels(int channels)

This function checks if c*hannels* contains a valid and acceptable value for the number of channels. If it is, this function changes the number of channels, stored in *wfx,* to c*hannels* and returns true. Also, it updates the fields in *wfx* whose value depends on the number of channels. Otherwise, the current value in *wfx* is retained and the function returns false.

int GetBits()

This function returns the number of bits per sample, which is contained in *wfx*.

bool SetBits(int bits)

This function checks if *bits* contains a valid and acceptable value for the number of bits per sample. If it is, this function changes the number of bits per sample, stored in *wfx*, to *bits* and returns true. Also, it updates the fields in *wfx* whose value depends on the number of bits. Otherwise, the current value in *wfx* is retained and the function returns false.

int GetSamplesPerSecond()

This function returns the number of samples per second, which is contained in *wfx*.

bool SetSamplesPerSecond(int samplesPerSec)

This function checks if *samplesPerSec* contains a valid and acceptable value for the sampling rate, in Hz. If it is, this function changes the number of samples per second, stored in *wfx,* to *samplesPerSec* and returns true. Also, it updates the fields in *wfx* whose value depends on the number of samples per second. Otherwise, the current value in *wfx* is retained and the function returns false.

3.4. File Input Functions

The file input function is a private function that is used to read WAV files. It accepts an input stream as one argument, which is associated with the file.

bool Load(string filename)

This is a public function that loads the contents of the file associated with *filename*, if it is a WAV file, into the object. The audio samples are stored in *audioBuffer* and *raudioBuffer* while the other information are stored in *wfx*. The indicators *maxBufferCount* and *lastBufferCount* are also set to their appropriate values. If the input file is a WAV file and the loading is successful, it returns true. If the input file is not a WAV file or the loading failed, it returns false. If the input file format is invalid, then its contents are not loaded and the currently loaded audio is erased.

3.5. Friend CALLBACK Function

friend void CALLBACK waveOutProc(HWAVEOUT hWaveOut, UINT uMsg,
DWORD dwInstance, DWORD dwParam1, DWORD dwParam2)

This function is the callback procedure for audio playback. This is not a member of the WAV class but since it is a friend function, it is included the discussion of the WAV class. Because it needs access to a private variable in the WAV class, it was made a friend function. When this function receives the message *WOM_DONE*, through *uMsg*, which indicates that playback of the current buffer is done, it increments the value of the free block counter, *freeBlockCount*. It ignores the other messages that are sent to it.

4. The UserInterface Class[4]

The UserInterface class provides the functionalities needed for user interaction. It contains two WAV objects that provide forward and reverse playback functionalities and functions for the program's user interface.

5.1. Constructor and Destructor

UserInterface()

The constructor initializes the audio parameters to the default values..

~UserInterface()

The destructor was not explicitly defined; the provided default destructor is used for object destruction.

5.2. Objects and Data Types

int recChannels, int recBits, int recSamplesPerSecond

These are private members used to store the initial parameters. The values of these will change depending on the header of the loaded WAV file.

5.3. Functions

void PrintEnterScreen()

This is a public function that prints the opening screen of the program.

void PrintExitScreen()

This is a public function that prints the closing screen of the program.

[4] Reference 1

void PrintUserInterface()

This is a public function that prints the main menu. It calls the *PrintPlayerMenu()* to print the specific commands applicable only to the current mode.

char GetAndExecuteCommand()

This is a public function that waits for a key to be pressed. Once a key is pressed, it executes the task corresponding to the key, if such task is defined. Otherwise, it does nothing. It always returns the value of the key that was pressed.

void PrintPlayerMenu()

This is a private function that displays the available commands for the player.

void Player()

This is a private function that performs the tasks necessary for audio playback, including the display of the interface. It calls the WAV class function *Play()*.

void rPlayer()

This is a private function that performs the tasks necessary for reverse audio playback, including the display interface. It calls the WAV class function *rPlay()*.

void Loader()

This is a private function that performs the necessary tasks for loading an audio to the player, including the display of the interface. It calls the WAV class function *Load()*.

char GetYesNoAnswer()

This is is a private function that reads a key press and repeatedly does so until the key pressed is either 'Y' or 'N'.

string InputString()

This is a private function that reads a string from the user. It is used for inputting a string to ensure that the input has a valid form before being processed further.

5. User's Manual

5.1. CDROM Contents

- <CD root directory>\
  - index.html – *homepage of project website (redirector)*
  - software\
    - wav.h – *WAV class definition*
    - wav.cpp – *WAV class implementation*
    - userinterface.h – *UserInterface class definition*
    - userinterface.cpp – *UserInterface class implementation*
    - main.cpp – *main() function*
  - doc\
    - documentation.odt – *project documentation (.odt file)*
    - documentation.pdf – *project documentation (.pdf file)*
    - progress_reports\ – *contains progress reports of group members*
      - conanan\
      - suarez\
      - villarica\
    - proposal\ - *contains approved project proposal*
      - proposal.odt
  - html\ – *contains project website files*

5.2. Using the Software

When the program is run, an opening screen (Figure 5.1) is displayed. Press a key to enter and proceed to the main menu.

Figure 5.1. Opening Screen

On the main menu (Figure 5.2), the player commands are displayed: Load audio, Play audio, Play reverse audio.

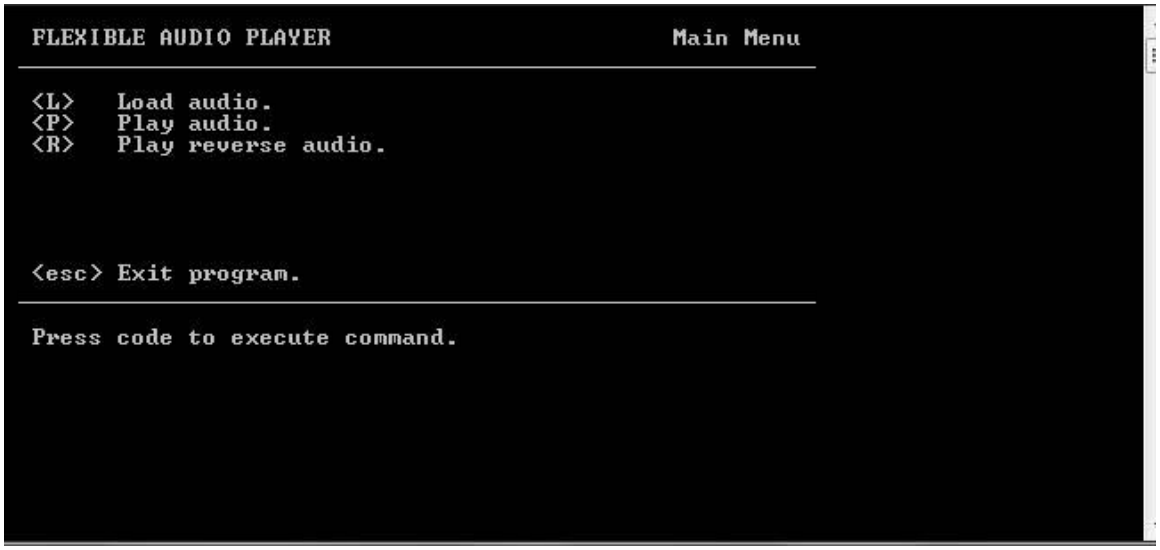

Figure 5.2. Main Menu

From the main menu, you may exit the program by pressing <esc>. This will display the closing screen (Figure 5.3) and then exits the program.

Figure 5.3. Closing Screen

5.2.1. Loading Audio

To load audio from a WAV file, press <L>. You will be prompted to enter a filename. Enter the complete filename of the WAV file you want to load. (Note: Filenames should not contain spaces) If an invalid filename is entered, you are given the option to enter another filename or abort the load operation and return to the main menu.

If the filename that you entered is not associated with an existing file or with a WAV file, no audio will be loaded.

The loading of the file may take some time. Avoid performing other commands until the result of the file loading operation is displayed.

Files to be loaded must be 8-bit, 16-bit, 24-bit, or 32-bit, single- or dual- channel audio, with sampling rates ranging from 8 kHz to 44.1 kHz.

5.2.2. Playing Loaded Audio

You are given the option to play the audio forwards or backwards. To play the loaded audio forwards, press <P>. To play the loaded audio in reverse, press <R>.

Anytime during forward or reverse playback, you have the option to pause the playback by pressing a key. Press a key again to resume playback.

After completion of forward or reverse playback, the result of the operation and the WAV audio information would be displayed.

6. Results and Recommendations

A Flexible Audio Player was not completely developed. The following improvement over the XMCS player was completed:

- Load WAV file into audio buffer

The group was able to complete the following:

- Load WAV file of any file size, and any bit rate (as long as it is a multiple of 8)
- Successfully reverse the audio samples for reverse playback
- Successfully open audio device

As an improvement over this project, it is recommended that the following features be revised:

- Play() and rPlay() functions
- Implement a pause function, that does not use iterators, using the waveOutPause and waveOutRestart functions

Appendix 1

Source Code

A1.1. WAV Class

A1.1.1. WAV Class Defintion: *wav.h*

```
#ifndef _WAV_H_
#define _WAV_H_

#include "utility.h"
#include <windows.h>
#include <mmsystem.h>
#include <fstream>
#include <string>
#include <list>
#include <conio.h>

using std::string;
using std::ifstream;
using std::ofstream;
using std::list;


class XMCS
{
    private:

        /*
         * to store audio format
         */
        WAVEFORMATEX wfx;

        /*
         * list for audio samples, and buffer size indicators
         */
        list<char*> audioBuffer;
        list<char*> raudioBuffer;
        int maxBufferSize;
        int lastBufferSize;
        int charPerSample=1;

        /*
        * declare iterators for pause, play and rplay
        */

        public list<char*>::iterator bli;
        public list<char*>::iterator ref;
        public list<char*>::iterator rbli;
        public list<char*>::iterator rref;
        public list<char*>::iterator pause_time;
        public list<char*>::iterator rpause_time;


        /*
        *  variable for checking the first run of either Play or rPlay
        *  set to 0 to indicate first run has happened
        */

        public int first_run = 0;
```

```
        /*
         * file radix, needed for reading and writing
         */
        int radix;

        /*
         * file input functions
         */
        char GetNextNonWhiteSpace(ifstream&);
        void DiscardUpToEol(ifstream&);
        short int GetNextSample(ifstream&);
        bool ReadFileHeader(ifstream&);
        int ReadBlockToBuffer(ifstream&, int, char*);


        /*
         * others, shared by functions (like global variables)
         */
        short int lineCharsRead;
        CRITICAL_SECTION criticalSection;
        volatile int freeBlockCount;

    public:

        /*
         * constructors, destructor
         */
        WAV();
        WAV(WAV&);
        ~WAV();

        /*
         * load audio samples from a file (filename) to both lists audioBuffer and
raudioBuffer
         */
        bool Load(string filename);

        /*
         * empty the list (audioBuffer and raudioBuffer)
         */
        bool Unload();

        /*
         * play audio samples in the list (audioBuffer)
         */
        bool Play();
        bool rPlay();

        /*
         * accessors
         */
        bool SetChannels(int);
        int  GetChannels();
        bool SetBits(int);
        int  GetBits();
        bool SetSamplesPerSecond(int);
        int  GetSamplesPerSecond();


        /*
         * callback procedure for playback
         */
        friend void CALLBACK waveOutProc(HWAVEOUT hWaveOut, UINT uMsg, DWORD dwInstance,
DWORD dwParam1, DWORD dwParam2);
};


#endif
```

A1.1.2. WAV Class Implementation: *wav.cpp*

```
#include "wav.h"

/*************************************************************************/
/* XMCS CLASS CONSTRUCTORS AND DESTRUCTOR                                */
/*************************************************************************/

WAV::WAV()
{
    // audio format
    wfx.nChannels       = 1;
    wfx.wBitsPerSample  = 16;
    wfx.nSamplesPerSec  = 22050;
    wfx.cbSize          = 0;
    wfx.wFormatTag      = WAVE_FORMAT_PCM;
    wfx.nBlockAlign     = (wfx.wBitsPerSample * wfx.nChannels) >> 3;
    wfx.nAvgBytesPerSec = wfx.nBlockAlign * wfx.nSamplesPerSec;

    // others
    radix               = 16;
    maxBufferSize       = 0;
    lastBufferSize      = 0;
}



WAV::WAV(WAV& wav)
{
    wfx             = wav.wfx;
    radix           = wav.radix;
    maxBufferSize   = wav.maxBufferSize;
    lastBufferSize  = wav.lastBufferSize;

    int  block_size = maxBufferSize;
    char *buffer;

    for (bli = wav.audioBuffer.begin() ; bli != wav.audioBuffer.end(); bli++)
    {
        ref = bli;
        if (++ref == wav.audioBuffer.end())
            block_size = lastBufferSize;
        buffer = new char[block_size];
        memcpy(buffer, *bli, block_size);
        audioBuffer.insert(audioBuffer.end(), buffer);
    }

    for (rbli = wav.raudioBuffer.begin() ; rbli != wav.raudioBuffer.end(); rbli++)
    {
        rref = rbli;
        if (++rref == wav.raudioBuffer.end())
            block_size = lastBufferSize;
        buffer = new char[block_size];
        memcpy(buffer, *rbli, block_size);
        raudioBuffer.insert(raudioBuffer.end(), buffer);
    }


}


WAV::~WAV()
{
    Unload();
}


/*************************************************************************/
/* WAV CLASS ACCESSOR FUNCTIONS                                          */
/*************************************************************************/
```

```
bool WAV::SetChannels(int val)
{
    if (!(val == 1 || val == 2))
        return false;

    wfx.nChannels       = val;
    wfx.nBlockAlign     = (wfx.wBitsPerSample * wfx.nChannels) >> 3;
    wfx.nAvgBytesPerSec = wfx.nBlockAlign * wfx.nSamplesPerSec;
    return true;
}



bool WAV::SetBits(int val)
{
    if (!(val == 8 || val == 16 ||  val == 24))
        return false;

    wfx.wBitsPerSample  = val;
    wfx.nBlockAlign     = (wfx.wBitsPerSample * wfx.nChannels) >> 3;
    wfx.nAvgBytesPerSec = wfx.nBlockAlign * wfx.nSamplesPerSec;
    return true;
}



bool WAV::SetSamplesPerSecond(int val)
{
    //if (!(val == 8000 || val == 11025 || val == 22050 || val == 44100))
    if (val < 8000 || val > 44100)
        return false;

    wfx.nSamplesPerSec  = val;
    //wfx.nBlockAlign     = (wfx.wBitsPerSample * wfx.nChannels) >> 3;
    wfx.nAvgBytesPerSec = wfx.nBlockAlign * wfx.nSamplesPerSec;
    return true;
}




int WAV::GetChannels()
{
    return wfx.nChannels;
}



int WAV::GetBits()
{
    return wfx.wBitsPerSample;
}



int WAV::GetSamplesPerSecond()
{
    return wfx.nSamplesPerSec;
}


/**************************************************************************/
/* WAV CLASS SPECIAL FUNCTIONS                                            */
/**************************************************************************/


bool WAV::Load(string filename)
{
```

```
const int BLOCK_SIZE = 1024;

char ChunkID[4], Format[4], Subchunk1ID[4],Subchunk2ID[4];
    int ChunkSize,Subchunk1Size, SampleRate, ByteRate,Subchunk2Size;
    short AudioFormat, NumChannels, BlockAlign, BitsPerSample;
    char** Data;
    char*  tempData;
    int lastBlockSize;
    int numBlocks;
    char *buffer;
    char *tempBuffer;
    int blockSize=1024;

    // Read the wave file header
    FILE *fhandle=fopen(filename.c_str(),"rb");
    fread(ChunkID,1,4,fhandle);
    fread(&ChunkSize,4,1,fhandle);
    fread(Format,1,4,fhandle);
    fread(Subchunk1ID,1,4,fhandle);
    fread(&Subchunk1Size,4,1,fhandle);
    fread(&AudioFormat,2,1,fhandle);
    fread(&NumChannels,2,1,fhandle);
    fread(&SampleRate,4,1,fhandle);
    fread(&ByteRate,4,1,fhandle);
    fread(&BlockAlign,2,1,fhandle);
    fread(&BitsPerSample,2,1,fhandle);
    fread(&Subchunk2ID,1,4,fhandle);
    fread(&Subchunk2Size,4,1,fhandle);
    lastBlockSize = Subchunk2Size%blockSize;
    charPerSample = BitsPerSample/8;
    if(lastBlockSize!=0)
      {
      numBlocks=(Subchunk2Size/blockSize)+1;
      }
else
      {
      numBlocks=Subchunk2Size/blockSize;
      }
lastBufferSize = lastBlockSize;
maxBufferSize = blockSize;

// dynamically allocate the array of blocks

Data = new char*[numBlocks];

// dynamically allocate each block array
for(int i=0; i<    numBlocks; i++)
      {
      Data[i]= new char[blockSize];
      }
// load audio data into the dynamic array Data by block
for(int i=0;i<numBlocks;i++)
      {
      tempData = new char[blockSize];
      fread(tempData,1,blockSize,fhandle);
      memcpy(Data[i],tempData,blockSize);
      }
fclose(fhandle);
// from the Data array, load the samples into the forward list, audioBuffer
for(int i=0;i<numBlocks;i++)
      {
      if(i==numBlocks-1)
      {blockSize = lastBlockSize;}
      tempBuffer= new char[blockSize];
      buffer = new char[blockSize];
      for(int j=0;j<blockSize;j++)
            {
            tempBuffer[j]=Data[i][j];
            }
      memcpy(buffer,tempBuffer,blockSize);
      audioBuffer.insert(audioBuffer.end(), buffer);
```

```
        }
    char *charBuffer;
    char *reverseBuffer[blockSize];
    int = reverse_counter=0;
    //load character pointers into reverse list

    for(int i = 0;i<numBlocks;i++)
        {

        if(i==numBlocks-1)
            {
            blockSize = lastBlockSize;
            }
        reverseBuffer = new char[blockSize];
        for(int j=0;j<blockSize;j++)
                {
                charBuffer = new char[charPerSample];
                charBuffer[reverse_counter]=Data[i][j];
                reverse_counter++;
                //counts if there have been charPerSample chars loaded
                if(reverse_counter==charPerSample)
                    {
                    //loads from charbuffer to reverseBuffer in correct order
                    for(int k=0;k<charPerSample;k++)
                        {
                        reverseBuffer[blockSize-j-1+k]=charBuffer[k];
                        }
                    reverse_counter=0;
                    }
                }
                //once reverseBuffer is full, it loads into the linked list raudioBuffer
        raudioBuffer.insert(raudioBuffer.end(), reversBuffer);
        }
     //free allocated memory
    delete [] tempBuffer;
    delete [] charBuffer;
    delete [] reverseBuffer[];
    delete []
    for( int i = 0 ; i < numBlocks ; i++ )
    delete [] Data[i];
    delete [] Data;
}



bool WAV::Unload()
{
    if (!audioBuffer.size())
        return false;

    int i = audioBuffer.size();
    int j = 0;

    while (audioBuffer.size() && maxBufferSize)
    {
        if (audioBuffer.size() != 1 || lastBufferSize)
            delete [] (*audioBuffer.begin());
        audioBuffer.erase(audioBuffer.begin());
    }
    if (!raudioBuffer.size())
        return false;

    int i = raudioBuffer.size();
    int j = 0;

    while (raudioBuffer.size() && maxBufferSize)
    {
        if (raudioBuffer.size() != 1 || lastBufferSize)
            delete [] (*raudioBuffer.begin());
        raudioBuffer.erase(raudioBuffer.begin());
    }
```

```
    lastBufferSize = maxBufferSize = 0;

    return true;
}

bool WAV::Play()
{
    const int BLOCK_COUNT   = 10;   // number of audio blocks for playback

    HWAVEOUT    hWaveOut;           // handle
    WAVEHDR     *waveBlocks;        // array of wave headers
    int         waveCurrentBlock;   // index of current block being played
    int         block_size;         // size of block to be played
    char        *buffer;            // temporary buffer


    // INITIALIZATION AND MEMORY ALLOCATION

    block_size = maxBufferSize;
    waveBlocks = new WAVEHDR[BLOCK_COUNT];
    for (int i = 0; i < BLOCK_COUNT; i++)
    {
        memset(&waveBlocks[i], 0, sizeof(WAVEHDR));
        waveBlocks[i].lpData = new char[block_size];
    }

    freeBlockCount  = BLOCK_COUNT;
    waveCurrentBlock    = 0;

    InitializeCriticalSection(&criticalSection);

    // PLAYBACK

    // open audio output device
    if(waveOutOpen(&hWaveOut, WAVE_MAPPER, &wfx, (DWORD_PTR)waveOutProc, (DWORD_PTR)this,
CALLBACK_FUNCTION) != MMSYSERR_NOERROR)
        return false;

    if(first_run==0)
         {
         pause_time = audioBuffer.begin();
         rpause_time = raudioBuffer.end();
         }
    bli = pause_time;
    rpause_time = rpause_time;

    buffer  = new char[block_size];

    // unless playback is stopped, play everything contained in the list
    while(bli != audioBuffer.end())
    {
        if (kbhit())
        {
            getch();
            waveOutReset(hWaveOut);
            pause_time = bli;
            rpause_time = rbli;
            bli = audioBuffer.end();
            break;
        }

        // copy data to be used to buffer
        memcpy(buffer, *bli, block_size);
        ref = bli++;

        // if the last element of the list, size may not be equal to the maximum
        if (++ref == audioBuffer.end())
            block_size = lastBufferSize;

        // unprepare header to be used for playback, in case not yet unprepared
        if(waveBlocks[waveCurrentBlock].dwFlags & WHDR_PREPARED)
```

```
            waveOutUnprepareHeader(hWaveOut,                &waveBlocks[waveCurrentBlock],
sizeof(WAVEHDR));

        // specify buffer length and audio samples
        waveBlocks[waveCurrentBlock].dwBufferLength = sizeof(char)*block_size;
        memcpy(waveBlocks[waveCurrentBlock].lpData, buffer, sizeof(char)*block_size);

        // prepare header to be used for playback
        if          (waveOutPrepareHeader(hWaveOut,          &waveBlocks[waveCurrentBlock],
sizeof(WAVEHDR)) != MMSYSERR_NOERROR)
            return false;

        // play
        waveOutWrite(hWaveOut, &waveBlocks[waveCurrentBlock], sizeof(WAVEHDR));

        // one less temporary buffer available
        EnterCriticalSection(&criticalSection);
        freeBlockCount--;
        LeaveCriticalSection(&criticalSection);

        // unless playback is stopped, wait until there's a free audio block
        while(!freeBlockCount)
        {
            if (kbhit())
            {
            getch();
            waveOutReset(hWaveOut);
            pause_time = bli;
            rpause_time = rbli;
            bli = audioBuffer.end();
            break;
            }
        }

        // current block is next block (imagine array is circular)
        waveCurrentBlock++;
        rbli--;
        waveCurrentBlock %= BLOCK_COUNT;
    }

    // unless playback is stopped, wait for all blocks to complete
    while(freeBlockCount < BLOCK_COUNT)
    {
        if (kbhit())
        {
            getch();
            waveOutReset(hWaveOut);
            break;
        }
    }

    // CLEAN-UP

    // unprepare any blocks that are still prepared
    for(int i = 0; i < freeBlockCount; i++)
        if(waveBlocks[i].dwFlags & WHDR_PREPARED)
            waveOutUnprepareHeader(hWaveOut, &waveBlocks[i], sizeof(WAVEHDR));

    DeleteCriticalSection(&criticalSection);
    waveOutClose(hWaveOut);

    delete [] buffer;

    for (int i = 0; i < BLOCK_COUNT; i++)
        delete [] waveBlocks[i].lpData;
    delete [] waveBlocks;
    first_run=1;
    return true;
}
bool WAV::rPlay()
{
```

```
    const int BLOCK_COUNT   = 10;   // number of audio blocks for playback

    HWAVEOUT    hWaveOut;           // handle
    WAVEHDR     *waveBlocks;        // array of wave headers
    int         waveCurrentBlock;   // index of current block being played
    int         block_size;         // size of block to be played
    char        *buffer;            // temporary buffer


    // INITIALIZATION AND MEMORY ALLOCATION

    block_size = maxBufferSize;
    waveBlocks = new WAVEHDR[BLOCK_COUNT];
    for (int i = 0; i < BLOCK_COUNT; i++)
    {
        memset(&waveBlocks[i], 0, sizeof(WAVEHDR));
        waveBlocks[i].lpData = new char[block_size];
    }

    freeBlockCount  = BLOCK_COUNT;
    waveCurrentBlock    = 0;

    InitializeCriticalSection(&criticalSection);

    // PLAYBACK

    // open audio output device
    if(waveOutOpen(&hWaveOut, WAVE_MAPPER, &wfx, (DWORD_PTR)waveOutProc, (DWORD_PTR)this,
CALLBACK_FUNCTION) != MMSYSERR_NOERROR)
        return false;

    if(first_run==0)
         {
         pause_time = audioBuffer.end();
         rpause_time = raudioBuffer.begin();
         }
    bli = pause_time;
    rpause_time = rpause_time;

    buffer  = new char[block_size];

    // unless playback is stopped, play everything contained in the list
    while(rbli != raudioBuffer.end())
    {
        if (kbhit())
        {
            getch();
            waveOutReset(hWaveOut);
            pause_time = bli;
            rpause_time = rbli;
            rbli = raudioBuffer.end();
            break;
        }

        // copy data to be used to buffer
        memcpy(buffer, *rbli, block_size);
        rref = rbli++;

        // if the last element of the list, size may not be equal to the maximum
        if (++rref == raudioBuffer.end())
            block_size = lastBufferSize;

        // unprepare header to be used for playback, in case not yet unprepared
        if(waveBlocks[waveCurrentBlock].dwFlags & WHDR_PREPARED)
            waveOutUnprepareHeader(hWaveOut,                &waveBlocks[waveCurrentBlock],
sizeof(WAVEHDR));

        // specify buffer length and audio samples
        waveBlocks[waveCurrentBlock].dwBufferLength = sizeof(char)*block_size;
        memcpy(waveBlocks[waveCurrentBlock].lpData, buffer, sizeof(char)*block_size);
```

```
        // prepare header to be used for playback
        if          (waveOutPrepareHeader(hWaveOut,          &waveBlocks[waveCurrentBlock],
sizeof(WAVEHDR)) != MMSYSERR_NOERROR)
            return false;

        // play
        waveOutWrite(hWaveOut, &waveBlocks[waveCurrentBlock], sizeof(WAVEHDR));

        // one less temporary buffer available
        EnterCriticalSection(&criticalSection);
        freeBlockCount--;
        LeaveCriticalSection(&criticalSection);

        // unless playback is stopped, wait until there's a free audio block
        while(!freeBlockCount)
        {
            if (kbhit())
            {
            getch();
            waveOutReset(hWaveOut);
            pause_time = bli;
            rpause_time = rbli;
            rbli = audioBuffer.end();
            break;
            }
        }

        // current block is next block (imagine array is circular)
        waveCurrentBlock++;
        bli--;
        waveCurrentBlock %= BLOCK_COUNT;
    }

    // unless playback is stopped, wait for all blocks to complete
    while(freeBlockCount < BLOCK_COUNT)
    {
        if (kbhit())
        {
            getch();
            waveOutReset(hWaveOut);
            break;
        }
    }

    // CLEAN-UP

    // unprepare any blocks that are still prepared
    for(int i = 0; i < freeBlockCount; i++)
        if(waveBlocks[i].dwFlags & WHDR_PREPARED)
            waveOutUnprepareHeader(hWaveOut, &waveBlocks[i], sizeof(WAVEHDR));

    DeleteCriticalSection(&criticalSection);
    waveOutClose(hWaveOut);

    delete [] buffer;

    for (int i = 0; i < BLOCK_COUNT; i++)
        delete [] waveBlocks[i].lpData;
    delete [] waveBlocks;
    first_run=1;
    return true;
}
/*****************************************************************************/
/* CALLBACK Procedure for WAV Audio Playback                                 */
/*****************************************************************************/



void CALLBACK waveOutProc(HWAVEOUT hWaveOut, UINT uMsg, DWORD dwInstance, DWORD dwParam1,
DWORD dwParam2)
{
```

```
    WAV* WAV_ptr = (WAV*) dwInstance;

    if(uMsg != WOM_DONE)
        return;

    EnterCriticalSection(&(WAV_ptr->criticalSection));
    (WAV_ptr->freeBlockCount)++;
    LeaveCriticalSection(&(WAV_ptr->criticalSection));
}
```

A1.2. UserInterface Class

A1.2.1. UserInterface Class Definition: *userinterface.h*

```
#ifndef _USERINTERFACE_H_
#define _USERINTERFACE_H_

#include "wav.h"
#include <string>
#include <conio.h>
#include <iostream>

using std::string;
using std::cout;
using std::cin;


class UserInterface
{
    private:

        XMCS Sound[2];
        bool state;

        int recChannels;
        int recBits;
        int recSamplesPerSecond;
        int recRadix;

        void PrintPlayerMenu();
        char GetYesNoAnswer();
        string InputString();

        void Player();
        void rPlayer();
        void Loader();

    public:

        UserInterface();
        void PrintEnterScreen();
        void PrintExitScreen();
        void PrintUserInterface();
        char GetAndExecuteCommand();
};



#endif
```

A1.2.2. UserInterface Class Implementation: *userinterface.cpp*

```
#include "userinterface.h"
#include <iostream>
#include <stdio.h>

using namespace std;


UserInterface::UserInterface()
{
    recChannels         = 1;
    recBits             = 8;
    recSamplesPerSecond = 11025;
    recRadix            = 16;
    state               = false;
}


void UserInterface::PrintEnterScreen()
{
    system("CLS");
    cout<<"                                                  \n"
        <<"                                                  \n"
        <<"                                                  \n"
        <<"    +++   +   +++      +      +++     +++     \n"
        <<"     ++  +++  ++      +++      +++   +++      \n"
        <<"      +++ + +++      ++ ++      +++ +++       \n"
        <<"       ++ + ++      ++ + ++      +++++       \n"
        <<"        +   +       ++   ++       +++        \n"
        <<"                                              \n"
        <<"                  P L A Y E R                 \n"
        <<"                                              \n"
        <<"                                              \n"
        <<"           Press a key to enter.";
    getch();
}



void UserInterface::PrintExitScreen()
{
    system("CLS");
    cout<<"                                                  \n"
        <<"                                                  \n"
        <<"                                                  \n"
        <<"    +++   +   +++      +      +++     +++     \n"
        <<"     ++  +++  ++      +++      +++   +++      \n"
        <<"      +++ + +++      ++ ++      +++ +++       \n"
        <<"       ++ + ++      ++ + ++      +++++       \n"
        <<"        +   +       ++   ++       +++        \n"
        <<"                                              \n"
        <<"                  P L A Y E R                 \n"
        <<"                                              \n"
        <<"                                              \n"
        <<"                Exiting...";
}



void UserInterface::PrintPlayerMenu()
{
    cout<<"  <L>   Load audio.                            \n"
        <<"  <P>   Play audio.                            \n"
        <<"  <R>   Play reverse audio.                    \n"
        <<"\n"
        <<"\n"
        <<"\n"
        <<"                                                \n";
}
```

```
void UserInterface::PrintUserInterface()
{
    system("CLS");
    cout<<"                                                          \n"
        <<"  FLEXIBLE AUDIO PLAYER                          Main Menu \n"
        <<" ____________________________________________________________\n"
        <<"                                                  \n";
        PrintPlayerMenu();
    cout<<"                                                  \n"
        <<"  <esc> Exit program.                             \n"
        <<" ____________________________________________________________\n"
        <<"                                                  \n"
        <<"  Press code to execute command. ";
}


/*FORWARD PLAYBACK PLAYER*/
void UserInterface::Player()
{
    char c;
    do
    {
        system("CLS");
        cout<<"                                                  \n"
            <<"  WAV PLAYER                                Play WAV audio.  \n"
            <<" ____________________________________________________________\n"
            <<"                                                  \n"
            <<"   Playing...                                     \n"
            <<"   Press a key to stop playing. ";

        if (Sound[state&1].Play())
            cout<<"\n"
                <<"                                              \n"
                <<"  ------------------------------              \n"
                <<"   XMCS Information                           \n"
                <<"                                              \n"
                <<"     channels           = "<<Sound[state&1].GetChannels()<<"\n"
                <<"     bits per sample    = "<<Sound[state&1].GetBits()<<"\n"
                <<"                                    samples       per       second       =
"<<Sound[state&1].GetSamplesPerSecond()<<"\n"
                <<"  ------------------------------              \n"
                <<"                                              \n"
                <<"   Playback successful.                       \n"
                <<"                                              \n";
        else
            cout<<"\n"
                <<"                                              \n"
                <<"   Playback failed.                           \n"
                <<"                                              \n";

        cout<<"   Repeat (Y/N)? ";
        c = GetYesNoAnswer();


    } while (toupper(c) == 'Y');
}


/*REVERSE PLAYBACK PLAYER*/
void UserInterface::rPlayer()
{
    char c;
    do
    {
        system("CLS");
        cout<<"                                                  \n"
            <<"  WAV PLAYER                                Play WAV audio.  \n"
            <<" ____________________________________________________________\n"
```

```
            <<"                                                  \n"
            <<"   Playing...                                     \n"
            <<"   Press a key to stop playing. ";

        if (Sound[state&1].rPlay())
            cout<<"\n"
                <<"                                              \n"
                <<"  ------------------------------              \n"
                <<"   XMCS Information                           \n"
                <<"                                              \n"
                <<"     channels           = "<<Sound[state&1].GetChannels()<<"\n"
                <<"     bits per sample    = "<<Sound[state&1].GetBits()<<"\n"
                <<"                                        samples       per       second       =
"<<Sound[state&1].GetSamplesPerSecond()<<"\n"
                <<"  ------------------------------              \n"
                <<"                                              \n"
                <<"   Playback successful.                       \n"
                <<"                                              \n";
        else
            cout<<"\n"
                <<"                                              \n"
                <<"   Playback failed.                           \n"
                <<"                                              \n";

        cout<<"   Repeat (Y/N)? ";
        c = GetYesNoAnswer();


    } while (toupper(c) == 'Y');
}


void UserInterface::Loader()
{
    string filename;
    char c;
    do
    {
        system("CLS");
        cout<<"                                                                  \n"
            <<"  FLEXIBLE AUDIO PLAYER       Load audio from WAV file.  \n"
            <<" ____________________________________________________________\n"
            <<"                                                              \n"
            <<"  Enter complete filename of WAV file: ";
        filename = InputString();

        c = 'N';

        if (filename == "")
        {
            cout<<"\n\n  Invalid filename."
                <<"\n\n  Enter another filename (Y/N)? ";

            c = GetYesNoAnswer();
        }

    } while (toupper(c) == 'Y');

    if (filename == "")
        cout<<"\n"
            <<"\n"
            <<"\n"
            <<"  Cancelled operation.";
    else
    {
        cout<<"\n"
            <<"  Loading audio. Please wait...";

        if (Sound[state&1].Load(filename))
            cout<<"\n"
                <<"\n"
```

```
                <<"  Loaded \""<<filename<<"\".";
        else
            cout<<"\n"
                <<"\n"
                <<"  Loading failed.";
    }
    getch();
}


char UserInterface::GetAndExecuteCommand()
{
    char c = getch();
    switch (toupper(c))
    {
        case 'P':
            Player();
            break;
        case 'R':
            rPlayer();
            break;
        case 'L':
            if (!state)
                Loader();
            break;
        default:
            return c;
            break;
    }
    return c;
}



string UserInterface::InputString()
{
    char buff[100000];
    cin.getline(buff, 100000, '\n');

    if (strlen(buff) > 99999)
    {
        return "";
    }
    else
    {
        for (int i = 0; i < strlen(buff); i++)
        {
            if (buff[i] == ' ')
                return "";
        }
    }
    return string(buff);
}



char UserInterface::GetYesNoAnswer()
{
    char c;
    do
    {
        c = getch();
    } while (toupper(c) != 'Y' && toupper(c) != 'N');

    return c;
}
```

A1.3. main() function: *main.cpp*

```
#include "userinterface.h"


int main(int argc, char *argv[])
{
    UserInterface UI;
    UI.PrintEnterScreen();
    do
    {
        UI.PrintUserInterface();
    } while (UI.GetAndExecuteCommand() != 0x1b);
    UI.PrintExitScreen();
    Sleep(1000);

    return EXIT_SUCCESS;
}
```

Bibliography

1. Baterina, Anna Veronica and Jeric Macalintal. “XMCS Player/Recorder.”

2. Davidson, Nathan. “How to Load a Wave File.” Internet. Available from http://www.cpp-home.com/tutorials/333_1.htm/, accessed 22 August 2010.

3. “Digital Audio – Creating a WAV (RIFF) file.” Internet. Available from http://www.topherlee.com/software/pcm-tut-wavformat.html/, accessed 30 August 2010.

4. Overton, David. “Playing Audio in Windows using waveOut Interface.” Internet. Available from http://www.downloadfreescript.com/source-code-detail-text/playing-audio-in-windows-using-waveout-interface-by-david-overton-in-graphics-sound-c-c.htm, accessed October 9, 2010.

5. Reisdorph, Kent. “Low-level wave audio, part 2.” Internet. Available from http://bcbjournal.org/articles/vol2/9808/Low-level_wave_audio__part_2.htm/, accessed 15 August 2010.

6. Wilson, Scott. “WAVE PCM soundfile format.” Internet. Available from https://ccrma.stanford.edu/courses/422/projects/WaveFormat/, accessed 8 August 2010.